Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–III EXAMINATION – SUMMER 2025

Subject Code:3130703 Date:31-05-2025

Subject Name:Database Management Systems

Time:02:30 PM TO 05:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			MARKS
Q.1*	**(a)**	Explain Logical & Physical data independence supported by DBMS	**03**
	(b)	Explain different types of users of DBMS.	**04**
	(c)	Compare database approach with traditional file systems to store application data.	**07**
Q.2	**(a)**	Explain Three Layer Schema Architecture of DBMS.	**03**
	(b)	Explain Specialization, Generalization and Categorization in EER Modeling.	**04**
	(c)	Explain Following Constraints supported by RDBMS: 1. Primary Key 2. Foreign Key / Referential Integrity Constraints 3. Entity Integrity 4. Domain Constraint	**07**
		OR	
	(c)	Explain Relational Algebra Operations in detail.	**07**
Q.3	**(a)**	Explain Recursive Relationship in ER Modeling with example.	**03**
	(b)	Explain Cardinality Ratio and Participation constraint of ER Model.	**04**
	(c)	What is the need to normalize data? Explain 1NF, 2NF & 3NF in detail.	**07**
		OR	
Q.3	**(a)**	Explain ACID Properties of transaction with appropriate example.	**03**
	(b)	Consider a relation R(A,B,C,D,E) with following dependencies: AB-→C, CD-→E, DE-→B . Is ABD a candidate key of this relation?	**04**
	(c)	Explain Inference Rules for Functional Dependency.	**07**
Q.4	**(a)**	What is the use of system log? What are the typical kinds of records in a system log? What are transaction commit points, and why are they important?	**03**
	(b)	Explain SQL Injection in brief.	**04**
	(c)	Explain Query Optimization with example.	**07**
		OR	
Q.4	**(a)**	Explain the use of Btrees.	**03**
	(b)	Explain Cursor in PL/SQL with example.	**04**
	(c)	Explain Conflict Serializability with precedence graph in Transaction Processing.	**07**
Q.5	**(a)**	Draw a state diagram, and discuss the typical states that a transaction goes through during execution.	**03**

(b) Explain Two phase locking protocol for guaranteeing Serializability. **04**

(c) Explain Deadlock handling in Transaction Processing. **07**

OR

Q.5 **(a)** Explain the use of group by and having clause in SQL queries. **03**

(b) Explain Triggers in PL/SQL with example. **04**

(c) Consider Following 3 Tables and Write SQL Queries. **07**

1. Books (BookID, BookTitle, Price, Author, Publisher)
2. Students (StudID, StudName, DOB, Gender, Branch, Sem)
3. Issue_Books (StudID, BookID, Issue_Date)

Query1: List all Books whose price in range of 300 to 500 Rs.
Query2: Display all Publisher Name & Total count of Books of that publisher.
Query3: Display list of all books which are not issued to any students.
Query4. Display the name students who are issued books in current month.
Query5: Display all Books assigned to student with name “mahesh”.
Query6: Display total no of books in library
Query7: Display the list of girl students who have taken book from library.

****

Q.1* (a) Explain Logical & Physical data independence supported by DBMS(3 mark)

Introduction / Definition:
Data independence refers to the ability of a **Database Management System (DBMS)** to change the database structure at one level **without affecting the other levels**. It is one of the most important features of DBMS and is supported by the **three-level database architecture**.

---

1) Logical Data Independence:

Logical data independence is the ability to **modify the conceptual (logical) schema** without changing the **external schemas or application programs**.

- Example: Adding a new attribute or table does not affect user views.
- It provides **flexibility in database design** and easier application maintenance.

---

2) Physical Data Independence:

Physical data independence is the ability to **change the internal (physical) storage structure** without affecting the **conceptual schema**.

- Example: Changing file organization or indexing method does not affect tables.
- It helps in improving **performance without rewriting programs**.

Key Points:

- Logical data independence is harder to achieve than physical data independence.
- DBMS ensures data independence using **schema mapping**.

(b) Explain different types of users of DBMS. (4 mark)

Introduction / Definition:

A **Database Management System (DBMS)** supports different categories of users who interact with the database according to their **roles, responsibilities, and access levels**. Each type of user performs specific functions to ensure efficient database usage and management.

---

1) Database Administrator (DBA):

- Responsible for **overall management** of the database.
- Defines **database schema, storage structures, and access permissions**.
- Performs **backup, recovery, and security control**.

---

2) Application Programmers (Developers):

- Write **application programs** that access the database using **SQL or APIs**.
- Design **forms, reports, and interfaces** for end users.
- Ensure proper implementation of business logic.

---

3) End Users:

End users interact with the database for data retrieval and updates. They are of the following types:

- **Naive (Parametric) Users:** Use predefined applications (e.g., bank clerks).
- **Casual Users:** Use occasional queries (e.g., managers).
- **Sophisticated Users:** Use complex queries and tools (e.g., analysts).

---

4) Database Designers:

- Design the **logical and physical structure** of the database.
- Identify **data requirements, relationships, and constraints**.
- Work closely with DBA and developers.

Key Points:

- Each DBMS user has a **specific role**.
- Proper user classification improves **security and efficiency**.

(c) Compare database approach with traditional file systems to store application data. (7 mark)

Introduction:

In early computer systems, data was stored using **traditional file-based systems**, where each application maintained its own data files. With the growth of data and applications, this approach caused several problems. The **database approach**, using a **DBMS**, was introduced to overcome the limitations of file systems by providing **centralized, integrated, and controlled data management**.

Traditional File System Approach:

- Data is stored in **separate files** for each application.
- Each application program defines its **own data structure**.
- There is **high data redundancy** due to duplicate data in multiple files.
- Data access is **program-dependent**.
- Data integrity and security are **hard to maintain**.

Database Approach:

- Data is stored in a **centralized database** managed by DBMS.
- Data structure is defined using **schemas**.
- **Minimal data redundancy** due to data sharing.
- Data access is **independent of application programs**.
- Provides strong **security, integrity, and concurrency control**.

Detailed Comparison Table:

Basis	**Traditional File System**	**Database Approach**
Data Redundancy	High	Low
Data Consistency	Poor	High
Data Independence	Not supported	Supported
Security	Weak	Strong
Data Sharing	Limited	Easy

Basis	Traditional File System	Database Approach
Backup & Recovery	Manual and difficult	Automatic and efficient
Concurrency Control	Not supported	Supported

Example in Words:

- In a file system, student data is stored in separate files for attendance, marks, and fees, causing duplication.
- In DBMS, all student data is stored in **one integrated database**, reducing redundancy.

Advantages of Database Approach:

- Improved **data consistency and integrity**.
- Easy **data access and modification**.
- Better **security and multi-user support**.

Q.2 (a) Explain Three Layer Schema Architecture of DBMS. (3 mark)

Introduction / Definition:

The **Three Layer Schema Architecture** of DBMS is a framework proposed by ANSI/SPARC to separate the database system into **three levels of abstraction**. The main objective of this architecture is to provide **data independence** and hide the complexity of database storage from users.

---

1) External Level (View Level):

- This level represents the **user's view** of the database.
- Each user can have a **different view** based on requirements.
- It hides unnecessary data and improves **security**.

---

2) Conceptual Level (Logical Level):

- Describes the **overall logical structure** of the entire database.
- Defines **tables, attributes, relationships, and constraints**.
- Acts as a bridge between external and internal levels.

---

3) Internal Level (Physical Level):

- Describes **how data is physically stored** in memory.
- Includes file organization, indexing, and storage structures.
- Focuses on **storage efficiency and performance**.

Key Points:

- This architecture supports **logical and physical data independence**.
- It improves **security, flexibility, and maintainability** of the database.

(b) Explain Specialization, Generalization and Categorization in EER Modelling. (4 mark)

Introduction:

The **Enhanced Entity Relationship (EER) model** extends the basic ER model by providing advanced modeling concepts such as

specialization, generalization, and categorization. These concepts help represent **real-world relationships** more accurately.

---

1) Specialization:

- Specialization is a **top-down approach** in which an entity set is divided into **sub-entity sets** based on distinguishing characteristics.
- Subclasses inherit **attributes and relationships** of the superclass.
- Example: *Employee* specialized into *Manager* and *Clerk*.

---

2) Generalization:

- Generalization is a **bottom-up approach** that combines **similar entity sets** into a **higher-level entity set**.
- Common attributes are moved to the generalized entity.
- Example: *Car* and *Truck* generalized into *Vehicle*.

---

3) Categorization (Union Type):

- Categorization is used when a subclass represents a **union of multiple entity sets**.
- It allows a single entity to belong to **more than one superclass**.
- Example: *Owner* can be a *Person* or a *Company*.

Key Points:

- Specialization and generalization support **inheritance**.

- Categorization is useful for modeling **complex relationships**.

(c) Explain Following Constraints supported by RDBMS: 1. Primary Key 2. Foreign Key / Referential Integrity Constraints 3. Entity Integrity 4. Domain Constraint (7 mark)

Introduction:

In a **Relational Database Management System (RDBMS)**, **constraints** are rules applied on database tables to ensure the **accuracy, consistency, and integrity** of data. These constraints restrict invalid data entry and help maintain reliable relationships among tables.

---

1) Primary Key Constraint:

- A **primary key** is an attribute or set of attributes that **uniquely identifies each tuple (row)** in a table.
- It **cannot contain NULL values** and must be **unique**.
- Only **one primary key** is allowed per table.
- It ensures **entity integrity**.
- Example in words: *Student_ID* uniquely identifies each student record.

---

2) Foreign Key / Referential Integrity Constraint:

- A **foreign key** is an attribute in one table that **refers to the primary key of another table**.
- It establishes a **relationship between two tables**.
- Referential integrity ensures that a foreign key value **must match an existing primary key value** or be NULL.

- Prevents **orphan records**.
- Example: *Dept_ID* in *Employee* table refers to *Department* table.

---

3) Entity Integrity Constraint:

- Entity integrity states that **primary key attributes cannot be NULL**.
- Every table must have a **unique identifier**.
- Ensures that each record is **distinct and identifiable**.
- Closely related to primary key constraint.
- Example: Student roll number cannot be NULL.

---

4) Domain Constraint:

- Domain constraint defines the **valid set of values** that an attribute can take.
- It includes **data type, size, format, and range**.
- Prevents invalid data entry.
- Example: Age must be between 18 and 60; Gender can be ‘M’ or ‘F’.

Advantages of Constraints:

- Maintain **data integrity and consistency**
- Improve **data reliability**
- Reduce errors during data insertion and update

OR

(c) Explain Relational Algebra Operations in detail. (7 mark)

Relational Algebra Operations

Introduction:

Relational Algebra is a **procedural query language** used in relational database systems to retrieve and manipulate data. It provides a **formal foundation** for relational databases and defines a set of **operations** that take one or more relations as input and produce a **new relation** as output.

Basic Relational Algebra Operations

1) Selection (σ):

- The **selection operation** selects tuples (rows) from a relation that satisfy a given condition.
- It is a **unary operation**.
- It does not change the number of attributes.
- Example in words: Select students whose marks are greater than 60.

---

2) Projection (π):

- Projection selects **specific attributes (columns)** from a relation.
- It removes duplicate tuples automatically.
- It is a **unary operation**.
- Example: Display only *Name* and *Roll_No* from Student table.

---

3) Union (∪):

- Union combines tuples from **two relations** with the **same structure**.
- Both relations must be **union compatible**.
- Duplicate tuples are eliminated.
- Example: Combine students from two different classes.

---

4) Set Difference (−):

- Returns tuples that are present in the **first relation but not in the second**.
- Relations must be union compatible.
- Example: Students who have not paid fees.

---

5) Cartesian Product (×):

- Combines **every tuple of one relation with every tuple of another relation**.
- Number of resulting tuples is the product of tuple counts.
- Used as a base for join operations.

Derived Relational Algebra Operations

6) Join (⋈):

- Join combines tuples from two relations based on a **common attribute or condition**.
- Types include **Theta Join, Equi Join, and Natural Join**.
- Example: Join Student and Department tables on Dept_ID.

---

7) Division (÷):

- Division is used when a query involves the **"for all" condition**.
- Returns tuples related to **all values in another relation**.
- Example: Students enrolled in all courses.

Key Features of Relational Algebra:

- Closed under operations (output is always a relation)
- Supports **query optimization**
- Provides theoretical foundation for SQL

Q.3 (a) Explain Recursive Relationship in ER Modeling with example. (3 mark)

Introduction / Definition:

A **recursive relationship** (also called a **unary relationship**) in **ER modeling** occurs when an **entity set is related to itself**. In this type of relationship, the **same entity participates more than once** with different roles.

Explanation:

- Recursive relationships are used to represent **hierarchical or self-referencing structures**.
- Each participating role is given a **role name** to avoid ambiguity.
- Cardinality constraints (one-to-one, one-to-many, many-to-many) can be applied.

Example:

- An **Employee** entity can have a relationship **"manages"** with itself.
- One employee acts as a **manager**, and another acts as a **subordinate**.
- This is commonly used to model **organizational hierarchies**.

Key Points:

- In ER diagram, the relationship connects **the same entity set**.
- Useful for representing **tree-like structures** in databases.

(b) Explain Cardinality Ratio and Participation constraint of ER Model. (4 mark)

In an ER Model, **Cardinality Ratio** defines the *maximum* number of entity instances involved in a relationship (e.g., One-to-One, One-to-Many, Many-to-Many), while **Participation Constraint** specifies the *minimum* number (Total or Partial), indicating if an entity *must* participate (Total) or *can* opt out (Partial) of a relationship, ensuring data integrity by enforcing structural rules.

1. Cardinality Ratio (Mapping Cardinality)

- **Definition:** Specifies the *maximum* number of relationship instances an entity can be associated with.
- **Types:**
  - **One-to-One (1:1):** One entity instance relates to at least one, but at most one, instance of another entity (e.g., One Employee manages One Department).
  - **One-to-Many (1:N):** One entity instance relates to many instances of another (e.g., One Department has Many Employees).

- **Many-to-One (N:1):** Many instances of one entity relate to one instance of another (e.g., Many Students enroll in One Course).
- **Many-to-Many (M:N):** Many instances of one entity relate to many instances of another (e.g., Many Students enroll in Many Courses).

2. Participation Constraint

- **Definition:** Specifies the *minimum* number of relationship instances an entity must participate in.
- **Types:**
  - **Total Participation (Mandatory):** Every instance of the entity *must* participate in the relationship (represented by a double line or '1' in min-max notation) (e.g., Every Course must be taught by a Faculty member).
  - **Partial Participation (Optional):** Some instances of the entity may *not* participate in the relationship (represented by a single line or '0' in min-max notation) (e.g., A Faculty member may not teach any Course).

Example (Student-Course)

- **Cardinality:** Many Students enrol in Many Courses (M:N).
- **Participation:** Students have partial participation (some students might not be enrolled yet), while Courses have total participation (every course must have a student enrolled).

(c) What is the need to normalize data? Explain 1NF, 2NF & 3NF in detail. (7 mark)

Introduction / Need for Normalization:

Normalization is a systematic process of organizing data in a database to **reduce redundancy and dependency**. The main

objective of normalization is to **eliminate data anomalies** such as insertion, deletion, and update anomalies, and to improve **data integrity and consistency** in relational databases.

Need for Normalization:

- To **reduce data redundancy**
- To **avoid update anomalies**
- To **maintain data consistency**
- To **improve data integrity**
- To **simplify database design and maintenance**

First Normal Form (1NF):

A relation is said to be in **First Normal Form (1NF)** if:

- All attributes contain **atomic (indivisible) values**
- There are **no repeating groups or multivalued attributes**
- Each record can be **uniquely identified**

Example in words:
If a student table contains multiple phone numbers in one column, it violates 1NF. Separating phone numbers into individual rows brings the table into 1NF.

Second Normal Form (2NF):

A relation is in **Second Normal Form (2NF)** if:

- It is already in **1NF**

- All non-key attributes are **fully functionally dependent on the primary key**
- There is **no partial dependency**

Example in words:
In a table with a composite key (Student_ID, Course_ID), if Student_Name depends only on Student_ID, it violates 2NF. Removing partial dependency achieves 2NF.

Third Normal Form (3NF):

A relation is in **Third Normal Form (3NF)** if:

- It is already in **2NF**
- There is **no transitive dependency**
- Non-key attributes depend **only on the primary key**

Example in words:
If Student table stores Dept_Name dependent on Dept_ID, which depends on Student_ID, it violates 3NF. Separating department details achieves 3NF.

Advantages of Normalization:

- Eliminates data redundancy
- Reduces anomalies
- Improves data consistency
- Enhances database efficiency

OR

Q.3 (a) Explain ACID Properties of transaction with appropriate example. (3 mark)

Introduction / Definition:

In a **Database Management System (DBMS)**, a **transaction** is a sequence of operations performed as a **single logical unit of work**. The **ACID properties** ensure that database transactions are processed **reliably and correctly**, even in the presence of failures.

---

1) Atomicity:

- Atomicity ensures that a transaction is **all-or-nothing**.
- Either all operations of a transaction are completed, or none are applied.
- Example: In a fund transfer, debit and credit must both occur.

---

2) Consistency:

- Consistency ensures that a transaction moves the database from **one valid state to another**.
- All integrity constraints must be maintained.
- Example: Account balance should never be negative.

---

3) Isolation:

- Isolation ensures that **concurrent transactions do not interfere** with each other.
- Intermediate results of a transaction are not visible to others.
- Example: Two users updating the same account cannot see partial updates.

---

4) Durability:

- Durability ensures that once a transaction is committed, its effects are **permanently stored**.
- Data remains safe even after system failure.
- Example: Committed bank transaction is not lost after crash.

(b) Consider a relation R(A,B,C,D,E) with following dependencies: AB-→C, CD-→E, DE-→B . Is ABD a candidate key of this relation? (4 mark)

Finding Candidate Key(s)

Given:

Relation **R(A, B, C, D, E)**
Functional Dependencies:

- **AB → C**
- **CD → E**
- **DE → B**

Step 1: Identify attributes not appearing on RHS

- RHS attributes: **C, E, B**
- Attributes not on RHS: **A, D**

👉 Therefore, **A and D must be part of every candidate key**.

Step 2: Find closure of {A, D}

$$(A, D)^+ = \{A, D\}$$

No new attribute is derived, so add required attributes.

Step 3: Try (A, B, D)

- AB → C ⇒ C obtained
- CD → E ⇒ E obtained
- DE → B ⇒ B already present

$$(ABD)+=\{A,B,C,D,E\}$$

Step 4: Check minimality

- Removing **A** or **D** fails to determine all attributes
- Hence, the key is **minimal**

Final Answer:

✅ **Candidate Key = {A, B, D}**

(c) Explain Inference Rules for Functional Dependency. (7 mark)

Introduction:

In **Database Management Systems (DBMS)**, **functional dependencies (FDs)** are constraints that describe the relationship between attributes in a relation. **Inference rules**, also known as **Armstrong's Axioms**, are a set of rules used to **derive all implied functional dependencies** from a given set of dependencies. These rules help in **database normalization and design**.

Armstrong's Inference Rules

1) Reflexivity Rule:

- If **Y ⊆ X**, then **X → Y**.

- It states that a set of attributes functionally determines itself.
- Example in words: If (A, B) is an attribute set, then (A, B) → A.

---

2) Augmentation Rule:

- If **X → Y**, then **XZ → YZ** for any attribute set Z.
- Adding the same attribute to both sides preserves the dependency.
- Example: If A → B, then AC → BC.

---

3) Transitivity Rule:

- If **X → Y** and **Y → Z**, then **X → Z**.
- Shows indirect functional dependency.
- Example: If Student_ID → Dept_ID and Dept_ID → Dept_Name, then Student_ID → Dept_Name.

Additional (Derived) Inference Rules

4) Union Rule:

- If **X → Y** and **X → Z**, then **X → YZ**.
- Combines dependencies with same left-hand side.

---

5) Decomposition Rule:

- If **X → YZ**, then **X → Y** and **X → Z**.
- Breaks complex dependency into simpler ones.

---

6) Pseudotransitivity Rule:

- If **X → Y** and **WY → Z**, then **WX → Z**.
- Used in complex dependency derivations.

Importance of Inference Rules:

- Help in finding **closure of attribute sets**
- Used in **normalization** and **candidate key identification**
- Ensure **consistency** in database design

Q.4 (a) What is the use of system log? What are the typical kinds of records in a system log? What are transaction commit points, and why are they important? (3 mark)

Introduction / Definition:

A **system log** (also called a **transaction log**) is a special file maintained by the **DBMS** that records all database modifications. It is mainly used for **recovery and maintaining database consistency** in case of system failure.

Use of System Log:

- Helps in **recovering the database** after system crash or failure
- Maintains a record of **all transactions and their operations**
- Supports **rollback and roll-forward recovery mechanisms**

Typical Records in a System Log:

- **Transaction Start record** – indicates beginning of a transaction
- **Update record** – contains old and new values of data

- **Commit record** – shows successful completion of transaction
- **Abort record** – indicates transaction failure

Transaction Commit Points:

A **commit point** is the moment when a transaction is **successfully completed** and all its changes are permanently saved in the database. Commit points are important because they ensure **durability and consistency**, and only committed transactions are **redone during recovery**.

(b) Explain SQL Injection in brief. (4 mark)

SQL Injection

Introduction / Definition:

SQL Injection is a type of **security attack** in which a malicious user inserts or injects **malicious SQL code** into an application's input fields. This causes the SQL query to behave unexpectedly, allowing the attacker to **access, modify, or delete database data** without authorization.

How SQL Injection Occurs:

- Occurs when **user input is directly concatenated** into SQL queries.
- Input validation is **not properly implemented**.
- The database executes the injected SQL commands as valid queries.

Example (in words):

If a login query checks username and password, an attacker can enter a condition like *always true*, causing the system to **bypass authentication**.

Effects / Risks:

- Unauthorized access to sensitive data
- Data manipulation or deletion
- Database structure disclosure

Prevention Methods:

- Use **prepared statements and parameterized queries**
- Apply **input validation**
- Limit database user privileges

(c) Explain Query Optimization with example. (7 mark)

Query optimization is the process a database management system (DBMS) uses to select the most efficient **execution plan** for a given SQL query. The goal is to minimize resource consumption (CPU, memory, disk I/O) and reduce query response time by considering various execution strategies and choosing the one with the lowest estimated cost.

Explanation

When a user submits an SQL query, which is a non-procedural language (meaning it specifies *what* to retrieve, not *how*), the DBMS query optimizer automatically analyzes potential access methods and join orders. It estimates the cost of each alternative plan based on internal statistics (like the number of rows in a table or the presence of indexes) and selects the optimal plan.

Key aspects of the process include:

- **Equivalence Rules**: The optimizer applies relational algebra equivalence rules to transform the original query into various equivalent forms. For example, applying selection operations earlier can significantly reduce the amount of data processed in subsequent operations like joins.
- **Cost Estimation**: Each potential execution plan is assigned a numerical cost based on expected resource usage (I/O, CPU, etc.) and cardinality (estimated number of rows returned).
- **Plan Selection**: The plan with the lowest overall cost is chosen for execution.

Example

Consider two tables, Employees (with millions of rows) and Departments (with a few hundred rows), and the following query to find employees in the 'Sales' department:

```
sql

SELECT E.Name

FROM Employees E, Departments D

WHERE E.DeptID = D.DeptID AND D.DeptName = 'Sales';
```

A naive approach to this query might involve:

1. **Plan A (Inefficient):** Perform a full **cross-product** (Cartesian join) of all rows in both Employees and Departments tables, creating a massive intermediate table. Then, filter this large intermediate table for the conditions E.DeptID = D.DeptID and D.DeptName = 'Sales'. This is extremely slow and resource-intensive.

2. **Plan B (Optimized):** The query optimizer rewrites the plan using an optimization heuristic: "perform selections before joins".

1. First, the optimizer filters the small Departments table to find the specific DeptID for 'Sales'. This yields only a few rows.

2. Then, it uses that specific DeptID to look up matching records in the large Employees table, potentially using an **index** on the DeptID column for fast retrieval. This avoids scanning the entire Employees table and the costly initial cross-product.

The optimizer determines that Plan B has a much lower cost than Plan A due to reduced I/O and CPU usage, thus selecting it for execution and resulting in a faster response time for the user.

OR

Q.4 (a) Explain the use of Btrees. (3 mark)

Introduction / Definition:
A B-Tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. It is a generalization of a binary search tree in which a node can have more than two children. B-Trees are widely used in databases and file systems where large amounts of data need to be stored and retrieved efficiently.

Detailed Explanation / Uses:

- B-Trees are primarily used to **store data on disk** because they minimize the number of disk reads.
- They are ideal for systems that read and write **large blocks of data**.
- A B-Tree keeps the tree **balanced**, so the height of the tree remains low, ensuring **faster search, insertion, and deletion**.

- Each node in a B-Tree can contain multiple keys, which reduces the tree height compared to a binary search tree.
- They are particularly useful in **database indexing**, where quick access to records is required.
- B-Trees also help in **file systems** like NTFS, ReiserFS, and Btrfs, where directories and file blocks are efficiently managed.

Advantages / Key Points:

- Guarantees **logarithmic time complexity** for search, insert, and delete operations.
- Efficient use of **disk storage** by reducing the number of I/O operations.
- Keeps data **sorted**, which simplifies range queries.
- Supports **dynamic growth** without significant reorganization.

Example (in words):
Consider a database of student records. Using a B-Tree to index student IDs allows the system to quickly locate a student's record without scanning the entire database, even if there are thousands of entries

(b) Explain Cursor in PL/SQL with example. (4 mark)

Introduction / Definition:
In PL/SQL, a **cursor** is a pointer that allows you to fetch and process **rows returned by a query one at a time**. Cursors are essential when a query returns **multiple rows**, and you want to process each row individually in a controlled manner. They act like a handle for the result set in memory.

Types of Cursors:

- **Implicit Cursor:** Automatically created by PL/SQL when a SELECT statement returns **only one row**.

- **Explicit Cursor:** Declared by the programmer for queries that return **multiple rows**, giving better control over row-by-row processing.

Features / Uses:

- Cursors help **traverse and manipulate** each row of a query result individually.
- They are widely used in **reports, data validation, and batch processing**.
- Provide **row-by-row processing**, unlike normal SQL statements that work on sets of rows.
- Help in **error handling** for queries returning multiple rows.

Steps to Use an Explicit Cursor:

1. **Declare the Cursor:** Specify the SQL query.
2. CURSOR emp_cursor IS SELECT emp_id, emp_name FROM employees;
3. **Open the Cursor:** Initialize it and execute the query.
4. OPEN emp_cursor;
5. **Fetch Rows:** Retrieve one row at a time into variables.
6. FETCH emp_cursor INTO v_emp_id, v_emp_name;
7. **Process Data:** Use the fetched data in your PL/SQL block (e.g., display or calculation).
8. **Close the Cursor:** Release the memory associated with the cursor.
9. CLOSE emp_cursor;

Example:

DECLARE

```
  CURSOR emp_cursor IS
    SELECT emp_id, emp_name FROM employees;
  v_emp_id employees.emp_id%TYPE;
  v_emp_name employees.emp_name%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_emp_id, v_emp_name;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_emp_id || ', Name: ' || v_emp_name);
  END LOOP;
  CLOSE emp_cursor;
END;
```

Explanation of Example:

- The cursor emp_cursor selects all employee IDs and names.
- The loop fetches each row one by one.
- DBMS_OUTPUT.PUT_LINE displays the data.
- %NOTFOUND ensures the loop exits when all rows are processed.

Advantages:

- Allows **row-by-row processing** for multi-row queries.
- Provides **better control** over query results.
- Supports **complex operations** like calculations or conditional processing per row

(c) Explain Conflict Serializability with precedence graph in Transaction Processing. (7 mark)

Introduction / Definition:
In transaction processing, **serializability** ensures that the concurrent execution of transactions produces the **same result** as if the transactions were executed **serially**, one after another. It is a key concept in **concurrency control**, which helps maintain **data consistency** in multi-user database systems. **Conflict serializability** is a type of serializability that examines **conflicting operations** among transactions to determine if a schedule can be transformed into a serial schedule without changing the final outcome.

Detailed Explanation:

- **Conflicting Operations:** Two operations are said to conflict if they meet all the following conditions:
    1. They belong to **different transactions**.
    2. They **access the same data item**.
    3. At least **one of them is a write operation**.
- Common types of conflicts:
    - **Read-Write (RW):** One transaction reads, and another writes the same data item.
    - **Write-Read (WR):** One transaction writes, and another reads the same data item.
    - **Write-Write (WW):** Both transactions write the same data item.
- **Conflict Serializability:** A schedule (sequence of operations) is **conflict serializable** if **swapping non-conflicting operations** can transform it into a serial schedule, ensuring **consistency of the database**.

Precedence Graph (Serialization Graph):

- A **precedence graph** is used to **test conflict serializability**.
- **Steps to Construct a Precedence Graph:**
    1. **Nodes:** Create a node for each transaction.
    2. **Edges:** Draw a directed edge from transaction Ti to Tj if an operation in Ti **conflicts** with a later operation in Tj on the same data item.
    3. **Cycle Detection:**
        - If the graph **has no cycles**, the schedule is conflict serializable.
        - If the graph **contains a cycle**, the schedule is **not conflict serializable**.

Key Features / Advantages:

- Ensures **consistency and correctness** in concurrent transaction execution.
- Provides a **systematic way** to check serializability using a graph.
- Supports **safe concurrency**, allowing multiple transactions to proceed without violating database integrity.

Disadvantages / Limitations:

- Constructing the graph can be **complex** for a large number of transactions.
- Only checks **conflict serializability**, not **view serializability**, which is a more general form.

Example (in words):
Consider two transactions:

- **T1:** Read(A), Write(A)
- **T2:** Read(A), Write(A)

Schedule: R1(A), R2(A), W1(A), W2(A)

- Conflicts: W1(A) conflicts with W2(A) and R2(A) conflicts with W1(A).
- Precedence graph:
  - Nodes: T1, T2
  - Edges: T1 → T2 (because W1 precedes W2), T2 → T1 (because R2 precedes W1)
- Since the graph **contains a cycle**, this schedule is **not conflict serializable**.

Applications:

- Used in **DBMS concurrency control mechanisms**.
- Ensures **data integrity** in **banking, reservations, and inventory systems**.
- Helps **database managers** decide **safe schedules** for transaction execution.

Q.5 (a) Draw a state diagram, and discuss the typical states that a transaction goes through during execution. (3 mark)

Introduction / Definition:
A **transaction** in a database system is a sequence of operations performed as a single logical unit of work, which must either **complete fully or have no effect at all**. Transactions go through different **states during execution**, reflecting their progress and status. Understanding these states is essential for **concurrency control, recovery, and reliability** in database management systems (DBMS).

Typical States of a Transaction:

1. **Active State:**
    - The transaction has **started execution** but has **not yet completed**.
    - All operations are being performed normally.
2. **Partially Committed State:**
    - The transaction has **executed its final operation** but the changes are **not yet permanently recorded** in the database.
    - DBMS ensures all updates are ready to be committed.
3. **Committed State:**
    - The transaction has **successfully completed**, and all changes are **permanently stored** in the database.
    - Guarantees **durability** of the transaction.
4. **Failed State:**
    - The transaction **cannot proceed** due to some **error** or system failure.
    - The changes made so far are **not yet rolled back**.
5. **Aborted State:**
    - The transaction has **rolled back**, undoing all changes made during execution.
    - The database is restored to its **previous consistent state**.

Explanation of the State Diagram:

- A **state diagram** shows the **transition of a transaction** between these states.

- The diagram begins at **Active**, moves to **Partially Committed**, then either to **Committed** if successful or to **Failed** and then **Aborted** in case of an error.
- Arrows indicate **possible transitions** between states. For example:
    - Active → Partially Committed → Committed
    - Active → Failed → Aborted

Purpose / Key Points:

- Helps DBMS **manage transaction life cycles** efficiently.
- Aids in **error handling and recovery**.
- Ensures **ACID properties**: Atomicity, Consistency, Isolation, Durability.

(b) Explain Two phase locking protocol for guaranteeing Serializability. (4 mark)

Introduction / Definition:
The **Two-Phase Locking (2PL) protocol** is a concurrency control method used in database systems to ensure **conflict serializability** of transactions. It regulates how transactions acquire and release **locks** on data items to prevent conflicts and maintain **database consistency** during concurrent execution. By following 2PL, a DBMS guarantees that the schedule of transactions is **serializable**, meaning it produces the same result as some serial execution of the transactions.

Explanation of Two Phases:
A transaction following the 2PL protocol goes through **two distinct phases**:

1. **Growing Phase:**
    - The transaction **acquires all the locks it needs** (shared/read or exclusive/write locks).

- No locks are **released** during this phase.
- Ensures that the transaction has **exclusive access to required resources** before proceeding.

2. **Shrinking Phase:**
   - The transaction **releases locks** after it has finished using the data items.
   - No new locks can be **acquired** in this phase.
   - Prevents conflicts and ensures **serializability**.

Types of Locks Used:

- **Shared Lock (S Lock):** Allows **multiple transactions** to read a data item but prevents writing.
- **Exclusive Lock (X Lock):** Allows a **single transaction** to write a data item, preventing others from reading or writing it.

Key Points / Advantages:

- Guarantees **conflict serializability** and maintains **database consistency**.
- Prevents **dirty reads** and ensures **isolation** between transactions.
- Can be applied in **distributed databases** as well.

Disadvantages / Limitations:

- May lead to **deadlocks**, where two or more transactions wait indefinitely for locks.
- Can reduce **concurrency** because transactions must hold locks until the shrinking phase.

Example (in words):

- Transaction T1 wants to read and write data item A.

- Transaction T2 wants to read data item A.
- Using 2PL, T1 first acquires an **exclusive lock** on A (growing phase).
- T2 must **wait** until T1 releases the lock (shrinking phase) to maintain serializability.

(c) Explain Deadlock handling in Transaction Processing. (7 mark)

Introduction / Definition:
In transaction processing, a **deadlock** is a situation where two or more transactions are **waiting indefinitely** for resources held by each other, preventing any of them from proceeding. Deadlocks are a critical problem in **concurrent database systems** because they can halt system operations, reduce performance, and violate **transaction atomicity and consistency**. Handling deadlocks effectively ensures smooth transaction processing and maintains **database integrity**.

Detailed Explanation:

- Deadlocks occur when the **following four conditions** hold simultaneously:
    1. **Mutual Exclusion:** Resources are non-shareable; only one transaction can use a resource at a time.
    2. **Hold and Wait:** A transaction is holding at least one resource while waiting for others.
    3. **No Preemption:** Resources cannot be forcibly taken from a transaction; they must be released voluntarily.
    4. **Circular Wait:** A circular chain of transactions exists, each waiting for a resource held by the next transaction in the chain.
- Deadlock handling in DBMS can be achieved through three main approaches:

1. Deadlock Prevention:

- Ensures that at least one of the four necessary conditions for deadlock **cannot hold**.
- Techniques:
  - **Mutual Exclusion:** Not always avoidable; some resources are inherently non-shareable.
  - **Hold and Wait Prevention:** Require transactions to request **all resources at once**.
  - **No Preemption:** Allow preemption of resources if a transaction cannot obtain all required locks.
  - **Circular Wait Prevention:** Impose a **total ordering** on resource types; transactions request resources in increasing order.
- **Advantage:** Deadlocks cannot occur.
- **Disadvantage:** May reduce system concurrency and efficiency.

2. Deadlock Detection and Recovery:

- Deadlocks are allowed to occur but are **periodically detected** using algorithms.
- **Precedence Graph Method:**
  - Construct a **wait-for graph** with transactions as nodes.
  - Draw an edge from Ti → Tj if Ti is waiting for a resource held by Tj.
  - **Cycle detection:** Presence of a cycle indicates a deadlock.
- **Recovery Methods:**
  - **Transaction Rollback:** Abort one or more transactions in the cycle to break the deadlock.

    - **Resource Preemption:** Take a resource from a transaction and assign it to another.
- **Advantage:** Maximizes concurrency; transactions proceed freely until deadlock occurs.
- **Disadvantage:** Detection algorithms consume system resources; rollback may cause loss of work.

3. Deadlock Avoidance:

- Transactions declare in advance the **maximum resources needed**.
- The DBMS only grants resources if the system remains in a **safe state** (no possibility of deadlock).
- **Banker's Algorithm** is a classical example of deadlock avoidance.
- **Advantage:** Avoids deadlocks dynamically.
- **Disadvantage:** Requires careful resource planning; may reduce throughput.

Example (in words):

- Transaction T1 holds **data item A** and requests **data item B**.
- Transaction T2 holds **data item B** and requests **data item A**.
- Both wait indefinitely for the other's resource, forming a **deadlock**.
- Using deadlock detection, the DBMS can **abort T2**, release its resources, allowing T1 to complete.

Key Points / Applications:

- Deadlock handling is crucial in **multi-user database systems, banking applications, reservation systems, and inventory management**.

- Helps maintain **ACID properties**, especially **consistency and atomicity**.
- Efficient handling improves **system performance** and prevents indefinite transaction blocking.

OR

Q.5 (a) Explain the use of group by and having clause in SQL queries. (3 mark)

Introduction / Definition:
In SQL, the **GROUP BY** and **HAVING** clauses are used to **aggregate data** and **filter grouped results**. They are essential for summarizing data from large tables and generating reports, making it easier to analyze patterns or trends in the database. These clauses are often used with **aggregate functions** like SUM(), COUNT(), AVG(), MIN(), and MAX().

GROUP BY Clause:

- The GROUP BY clause **groups rows** that have the same values in specified columns into **summary rows**.
- It is typically used with **aggregate functions** to perform calculations on each group.
- Example:
- SELECT department_id, COUNT(*) AS num_employees
- FROM employees
- GROUP BY department_id;
    - This query counts the number of employees in each department.
    - GROUP BY department_id groups the rows based on the department.

HAVING Clause:

- The HAVING clause is used to **filter groups** after aggregation, unlike the WHERE clause, which filters **individual rows** before aggregation.
- Example:
- SELECT department_id, COUNT(*) AS num_employees
- FROM employees
- GROUP BY department_id
- HAVING COUNT(*) > 5;
    - This query shows only departments with **more than 5 employees**.
    - HAVING applies the condition to the **grouped results**.

Purpose / Key Points:

- GROUP BY organizes data for **summary and analysis**.
- HAVING filters **aggregated results**, which cannot be done with WHERE.
- Together, they are useful for generating **reports, statistics, and business insights**.

(b) Explain Triggers in PL/SQL with example. (4 mark)

Introduction / Definition:
In PL/SQL, a **trigger** is a **stored procedure** that is automatically executed (or "triggered") by the database when a specific **event** occurs on a table or view. Triggers are widely used to **enforce business rules, maintain audit trails, and ensure data integrity** without requiring explicit action by the user.

Key Features / Uses:

- **Automatic Execution:** Runs automatically when a specified event occurs.
- **Event-Driven:** Triggered by events like INSERT, UPDATE, or DELETE.
- **Maintains Data Integrity:** Ensures rules are followed consistently.
- **Auditing:** Tracks changes in the database (e.g., logging modifications).
- **Complex Validations:** Can perform checks that are difficult to enforce with constraints alone.

Types of Triggers:

1. **Row-Level Trigger:** Executes once for **each affected row**.
2. **Statement-Level Trigger:** Executes **once per SQL statement**, regardless of the number of affected rows.

Example:

```
CREATE OR REPLACE TRIGGER emp_salary_check

BEFORE INSERT OR UPDATE ON employees

FOR EACH ROW

BEGIN

  IF :NEW.salary < 10000 THEN

    RAISE_APPLICATION_ERROR(-20001, 'Salary must be at least 10000');

  END IF;

END;
```

Explanation of Example:

- The trigger emp_salary_check is executed **before an INSERT or UPDATE** on the employees table.
- FOR EACH ROW makes it a **row-level trigger**.
- :NEW.salary refers to the **new value** being inserted or updated.
- If the salary is less than 10,000, the trigger raises an error, **preventing the operation**.

Advantages:

- Ensures **automatic enforcement** of rules.
- Reduces **manual coding** in applications.
- Can perform **complex validations and logging** automatically.

(c) Consider Following 3 Tables and Write SQL Queries.

1. Books (BookID, BookTitle, Price, Author, Publisher)

2. Students (StudID, StudName, DOB, Gender, Branch, Sem)

3. Issue_Books (StudID, BookID, Issue_Date)

Query1: List all Books whose price in range of 300 to 500 Rs.
Query2: Display all Publisher Name & Total count of Books of that publisher.

Query3: Display list of all books which are not issued to any students.
Query4. Display the name students who are issued books in current month.

Query5: Display all Books assigned to student with name “mahesh”.
Query6: Display total no of books in library Query7: Display the list of girl students who have taken book from library.

Introduction / Definition:
SQL (Structured Query Language) is used to **retrieve, manipulate, and manage data** stored in relational databases. The following queries demonstrate common operations like **selection, aggregation,**

joins, and subqueries on three tables: Books, Students, and Issue_Books.

Query 1: List all Books whose price is in the range of 300 to 500 rs.

```
SELECT BookID, BookTitle, Price, Author, Publisher
FROM Books
WHERE Price BETWEEN 300 AND 500;
```

- Uses the BETWEEN operator to **filter books by price**.

Query 2: Display all Publisher Names & Total count of Books of that publisher.

```
SELECT Publisher, COUNT(*) AS Total_Books
FROM Books
GROUP BY Publisher;
```

Uses GROUP BY to aggregate books by publisher.

COUNT(*) gives the total number of books per publisher.

Query 3: Display list of all books which are not issued to any students.

```
SELECT BookID, BookTitle
FROM Books
WHERE BookID NOT IN (SELECT BookID FROM Issue_Books);
```

- Uses a **subquery** to find books **not present in Issue_Books**.

Query 4: Display the names of students who are issued books in the current month.

```
SELECT DISTINCT S.StudName
FROM Students S
```

```
JOIN Issue_Books I ON S.StudID = I.StudID
WHERE EXTRACT(MONTH FROM I.Issue_Date) =
EXTRACT(MONTH FROM SYSDATE)
  AND EXTRACT(YEAR FROM I.Issue_Date) = EXTRACT(YEAR
FROM SYSDATE);
```

- Joins Students and Issue_Books to find **currently active issues**.
- EXTRACT(MONTH/YEAR) ensures only **current month issues** are selected.

Query 5: Display all Books assigned to student with name "mahesh".

```
SELECT B.BookID, B.BookTitle, B.Author, B.Publisher
FROM Books B
JOIN Issue_Books I ON B.BookID = I.BookID
JOIN Students S ON I.StudID = S.StudID
WHERE S.StudName = 'mahesh';
```

- Uses **JOINs** to find books issued specifically to student **mahesh**.

Query 6: Display total number of books in library.

```
SELECT COUNT(*) AS Total_Books
FROM Books;
```

- Simple aggregation using COUNT(*) to get **total books**.

Query 7: Display the list of girl students who have taken books from the library.

```
SELECT DISTINCT S.StudName
FROM Students S
JOIN Issue_Books I ON S.StudID = I.StudID
WHERE S.Gender = 'Female';
```

- Joins Students and Issue_Books and **filters female students**.
- DISTINCT ensures **no duplicate names** appear.

Explanation / Key Points:

- Queries demonstrate **filtering, aggregation, subqueries, and joins**.
- GROUP BY and COUNT are used for **summary information**.
- JOIN ensures **related data from multiple tables** can be combined.
- Subqueries and NOT IN are used to **identify unissued books**.
- EXTRACT function is used for **date-based filtering** (current month).